Core Java:
Object Oriented Programming II

Persistent Interactive | Persistent University

Key learning points

- At the end of this module, you will be able to understand :
  - Static data members and methods
  - Inheritance: Creating subclasses
  - Polymorphism : Method Overloading
  - Polymorphism : Method Overriding
  - Use of super
  - Constructors Chaining
  - Use of this() and super()
  - Creating Abstract Classes and Methods
  - Creating Interfaces and Methods
  - Reference Variable casting
  - Enums

Understanding static

- Static is a keyword.
- It is a non access modifier.
- Things you can mark as static:
  - Methods
  - Variables
  - Top-level nested classes
  - Block

Understanding static

- Static
  - belongs to the class not to any of it’s objects.
  - class need not be instantiated to access static members.

- static variables
  - are initialized (explicitly or to their default values) when the class is loaded.

- static methods
  - directly access other static members in the class.
  - cannot access instance members of the class.
  - can always use a reference of the class’s type to access its members (static or non-static).
  - Cannot be overridden

Accessing static Methods & Variables

- Non static fields from Static methods

```
class Frog
{
        int frogSize = 0;
        public int getFrogSize() {
        return frogSize;
        }
        public Frog(int s) {
         frogSize = s;
        }
        public static void main (String [] args) {
        Frog f = new Frog(25);
         //Accessing instance method
        System.out.println(f.getFrogSize());
        }
}
```

Accessing static Methods & Variables...

- static fields from Static methods

```
class Frog {
    // Declare and initialize static variable
    static int frogCount = 0;
    public Frog() {
    frogCount += 1; }// Modify the value in the constructor
  }
class TestFrog {
             public static void main (String [] args) {
              Frog f = new Frog();
             //using  classname
 System.out.print("frogCount:"+Frog.frogCount);
              //using dot operator
 System.out.print("frogCount:"+f.frogCount);
             }
}
```

Static block

- A block can be declared as static where the keyword static is used as follows.
- Static block gets executed before main method gets executed.
- So if you want some functionality, values to be available for your class even before main method is executed, those can be written in static block.

```
public class StaticBlockDemo
{
    static{
            //This is static block
     }
     public static void main(String args[])
     {
       //This is main method
     }
}
```

Things you can’t mark as static

- Constructors
- Classes
- Interfaces
- Local variables
- Inner classes
- Inner class methods and instance variables

Inheriting classes

- A Java class inherits another class using the extends keyword after the class name followed by the parents.

- The child class inherits all the instance variables and methods of the parent class

ChildClass extends ParentClass

```
class ParentClass
{
}
public class ChildClass extends ParentClass
{


}
```

Inheritance

- Creating a new class from the existing class.

Class Derived is implemented by extending already existing Base class

Class Cat is implemented by extending already existing Animal class

```
public class Base{   }

public class Derived extends Base{  }

public class Animal{   }

public class Cat extends Animal{  }
```

How constructors work in inheritance?

- Even though we are creating object of sub class super class constructor gets invoked first and then sub class constructor is invoked.
- This call is from sub class constructor to super class constructor.
- If there are default or non parametric constructors in both sub class and super class, the call to super class constructor is implicit.
- Call to parametric constructor from sub class constructor must be explicit.

super : Constructor Call

- super() is used in a subclass constructor to invoke a constructor in the immediate superclass.

- super invokes relevant constructor from superclass based on the signature of the call.

super : Constructor Call

```
public class Box {
       private double height;
       private double depth;
       Box(double height; double depth ){
           this.height = height;
           this.depth = depth ;
        }
}
// BoxWeight uses super to initialize its Box attributes.
class BoxWeight extends Box
{
       private double weight;  // weight of box
      // initialize height and depth using super()
       BoxWeight(double weight, double height, double
                depth) {
                 // call superclass constructor
                super(height, depth);
                this.weight = weight;
 }
}
```

super : Constructor Call…

// Compile time error

// Compile time error

```
class Animal {
        Animal(String name) { }
}
class Horse extends Animal {
        Horse() {
                super();
        }
}

class Clothing {
        Clothing(String s) { }
}
class TShirt extends Clothing { }
```

Polymorphism : Method overloading

- Java allows to define several methods with same name within a class
- Methods are identified with the parameters set based on:
  - the number of parameters
  - types of parameters
  - order of parameters.
- Compiler selects the proper method.
- This is called as Method overloading.

Polymorphism : Method overloading

- Method overloading is used to perform same task with different available data
- Example:

```
int sum(int a,int b) { return a+b;}

int sum(int a,int b,int c) { return a+b+c;}

float sum(float a, float b, float c) { return a+b+c;}
```

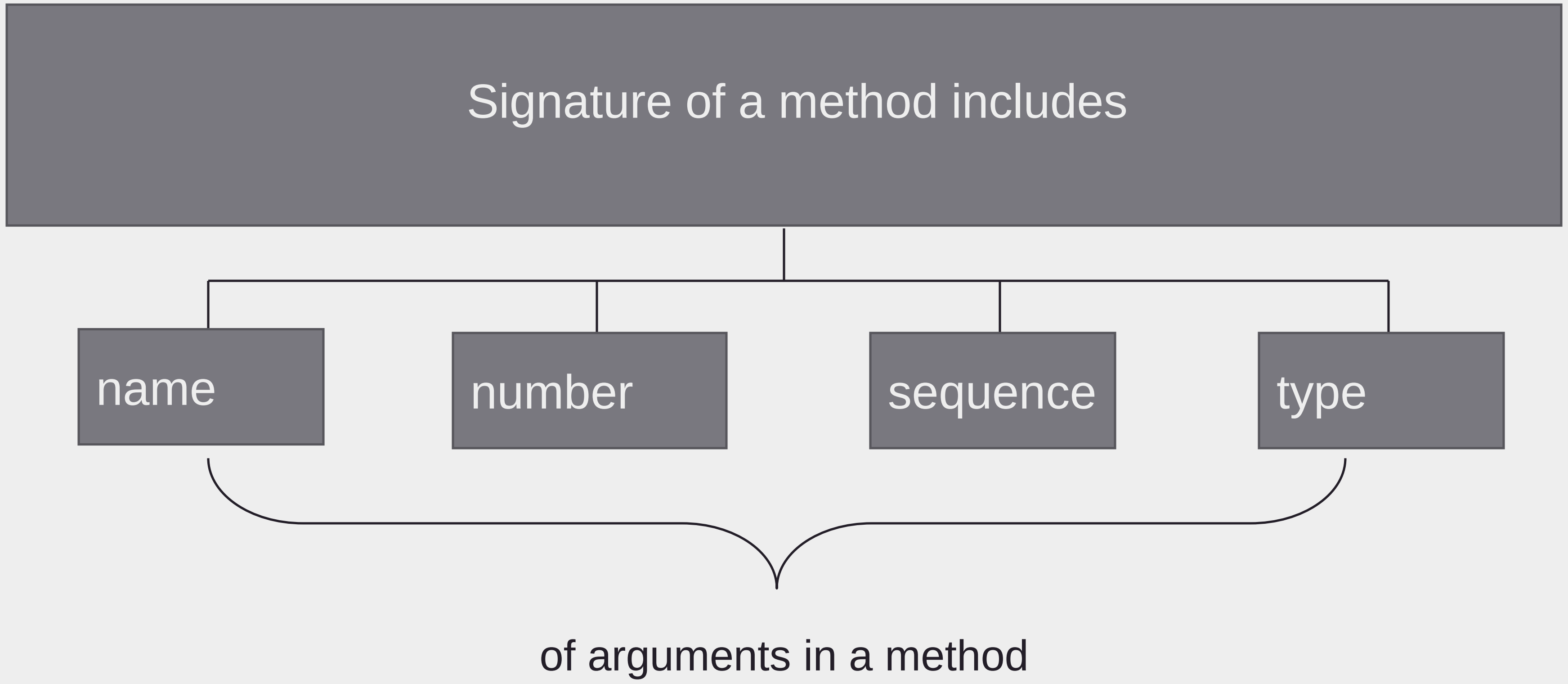
Polymorphism : Method Overloading

- Some Rules
  - Overloaded methods must change the argument list.
  - Overloaded methods can change the return type.
  - Overloaded methods can change the access modifier.
  - A method can be overloaded in the same class or in a subclass.

    (to be discussed later)
  - Overloaded methods can declare new or broader checked exceptions.

Polymorphism : Method Overriding

Method overriding is defined as creating a non-static method in the subclass that has the same return type and signature as a method defined in the super class.



Polymorphism : Method Overriding

```
class Animal {
    public void eat() {System.out.println("Generic Animal
Eating Generically");
}}
class Horse extends Animal {
   public void eat() {
   System.out.println("Horse eating hay, oats, and horse
               treats"); }
   public void buck() { }  }
 public class TestAnimals {
     public static void main (String [] args) {
     Animal animal = new Animal();
 // Runs the Animal version of eat()
animal.eat();
 //Animal ref, but a Horse object
animal = new Horse();
 // Runs the Horse version of eat()
                animal.eat();     }    }
```

Polymorphism : Method Overriding

- The compiler looks only at the reference type, not the instance type.
- The overriding method cannot have a more restrictive access modifier than the method being overridden.

```
Animal horse = new Horse();

horse.buck();


class Horse extends Animal {

      private void eat() {

         System.out.println("Horse eating hay, oats, and
horse treats");

}
```

// Can't invoke buck();
// Animal class doesn't have that method

// Compile time error

Polymorphism : Method Overriding

- The argument list must exactly match that of the overridden method.
- The return type must exactly match that of the overridden method.
- The access level must not be more restrictive than that of the overridden method.
- You cannot override the static and final methods of a superclass.

Polymorphism : Method Overriding

- The overriding method must not throw new or broader checked exceptions.

- The overriding method can throw narrower or fewer exceptions.

Hiding

- Hiding applies to members that would normally be inherited but are not because of a declaration of the same identifier in a subclass.
- A method can hide a method in the superclass by overriding it.
- Static methods are hidden.
- A static method in a subclass cannot hide an instance method in the superclass.

Hiding

```
class TestShadowing {
        static int x = 1;    // field variable
        public static void main(String[] args) {
                int x = 0;          // local variable
                System.out.println("x = " + x);
                System.out.println("TestShadowing.x
  = " +  TestShadowing.x);
        }}
```

super : Name Hiding

```
class A {     int i;     }

// Create a subclass by extending class A.

class B extends A {

        int i;                                  // this i hides the i in A

        B(int a, int b) {

                super.i = a;

                 i = b;     }

        void show() {

            System.out.println("i in superclass: " + super.i);

             System.out.println("i in subclass: " + i);}   }

class UseSuper {

                public static void main(String args[]) {

                                B subOb = new B(1, 2);
                                subOb.show();    }   }
```

Final classes: A way for preventing classes being extended

We can prevent an inheritance of classes by other classes by declaring them as final classes.
This is achieved in Java by using the keyword final as follows:

```
final class Regular
{ // members
}
 final class Employee extends Person
{     // members
}
```

Any attempt to inherit these classes will cause an error.

Final members: A way for preventing overriding of members in subclasses

- All methods and variables can be overridden by default in subclasses.
- This can be prevented by declaring them as final using the keyword “final” as a modifier. For example:

```
final void display();
```

- This ensures that functionality defined in this method cannot be altered any. Similarly, the value of a final variable cannot be altered.

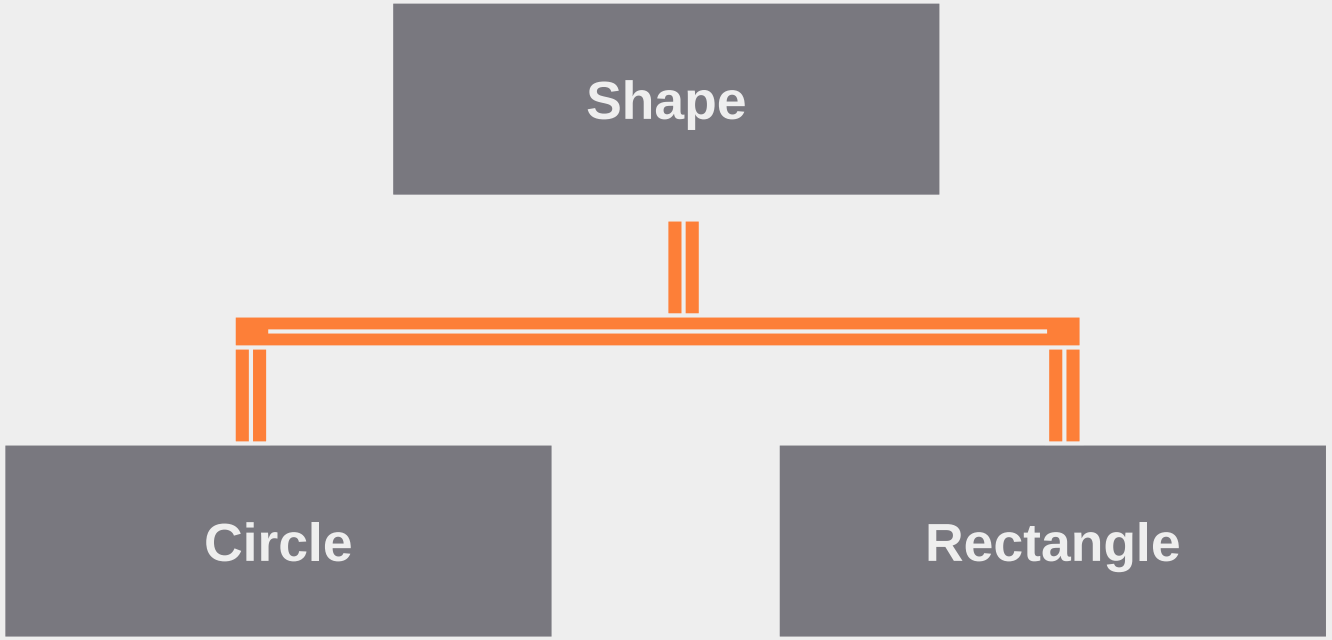
Understanding abstract class

- An Abstract class is a conceptual class.
- An Abstract class cannot be instantiated – objects cannot be created.
- Abstract classes provides a common root for a group of classes, nicely tied together in a package.
- When we define a class to be “final”, it cannot be extended. In certain situation, we want properties of classes to be always extended and used. Such classes are called Abstract Classes.

Properties of an abstract class

- A class with one or more abstract methods is automatically abstract and it cannot be instantiated.
- A class declared abstract, even with no abstract methods can not be instantiated.
- A subclass of an abstract class can be instantiated if it overrides all abstract methods by implementation them.
- A subclass that does not implement all of the superclass abstract methods is itself abstract; and it cannot be instantiated.
- We cannot declare abstract constructors or abstract static methods.

Creating abstract classes and methods



Persistent

Creating abstract classes and methods (continued)

Syntax

```
abstract class ClassName
{
                ...
                ...
                abstract DataType
                    MethodName1();
                ...
                ...
                DataType Method2()
                {
                    // method body
                }
}
```

Example

```
 abstract public  class Shape
 {

                    public abstract
double area();

                    public void move()
          { // non-abstract method
            // implementation
          }
}
```

Creating abstract classes and methods (continued)

```
 public Circle extends Shape {
            private double r;
            private static final double PI =3.1415926535;
            public Circle() { r = 1.0; }
            public double area() { return PI * r * r; }
   …
  }
 public Rectangle extends Shape {
            private double l, b;
            public Rectangle() { l = 0.0; b=0.0; }
            public double area() { return l * b; }
...
  }
```

Understanding interfaces

- A programmer’s tool for specifying certain behaviors that an object must have in order to be useful for some particular task.
- Interface is a conceptual entity.
- Can contain only constants (final variables) and non-implemented methods (Non – implemented methods are also known as abstract methods).

Creating interfaces and methods

Syntax to create an interface

```
interface InterfaceName
  {
            // Constant/Final Variable Declaration
            // Methods Declaration – only method body
  }
```

Implementing Driver interface

```
public interface Driver
   {
     void turnWheel (double angle);
     void applyBrake (double amount);
  }
```

Creating interfaces and methods (Java 8)

Interface syntax as per Java 8

Driver interface implementation as per Java 8

```
interface InterfaceName
  {
            // Constant/Final Variable Declaration
            // Methods Declaration – only method body
            //static implemented method
            //default implemented method
  }
public interface Driver
   {
     void  addDriver(Driver driver);
     static boolean isNull(String str) {
          if(str.equals(null))
          return true;
            }
    default void print( Driver driver) {
            System.out.println(driver);
     } }
```

Creating interfaces and methods (Java 9)

Interface declaration as per java 9

Driver interface implementation as per Java 9

```
interface InterfaceName
  {          // Constant/Final Variable Declaration
             // Methods Declaration – only method body
             //static implemented method
             //default implemented method
             //private method
            //private static method
  }
public interface Driver
   {
     void  addDriver(Driver driver);
    static boolean isNull(String str) { if(str.equals(null)
             return true; }
    default void print( Driver driver)    {
             System.out.println(driver); }
    private void log(String msg) {}
    private static void method() {}
  }
```

Creating interfaces and methods (continued)

- Interfaces are used like super-classes who properties are inherited by classes. This is achieved by creating a class that implements the give interface as follows:

```
public class BusDriver extends Person implements
Driver
   {
   // include each of the two methods from Driver
   }

class ClassName implements Interface1, Interface2,….,
InterfaceN
 {
             // Body of Class
  }
```

Inheriting interfaces

- Like classes, interfaces can also be extended. The new sub-interface will inherit all the members of the super-interface in the manner similar to classes.
- This is achieved by using the extends keyword.

```
interface InterfaceName2  extends  InterfaceName1
  {
              // Body of InterfaceName2
  }
```

Dynamic Method Dispatch

- is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time.
- a superclass reference variable can refer to a subclass object.
- uses concept of late-binding.

Polymorphism & Dynamic Method Lookup

- Polymorphism:
  - allows a reference to denote objects of different types at different times during execution.

- Dynamic Method Lookup :
  - process of determining the which object definition a method signature denotes during run-time.
  - based on the type of object.

Converting and Casting Object References

- Classes should be compatible otherwise compile time error.
- Casts that are permitted at compile-time include casting any object to its own class or to one of its sub or superclass types or interfaces.
- The compile-time rules cannot catch every invalid cast attempt. If the compile-time rules permit a cast, then additional, more stringent rules apply at runtime.

Casting with Object references

- A reference of any object can be cast to a reference of type Object
- If you cast up the class hierarchy you do not have to use the cast operator; if you are cast down the class hierarchy you must use the cast operator
- The compiler uses the declared type to verify the correctness of each method call. i.e. you cannot invoke a subclass method from a superclass reference.
- A cast may work at compile-time but fail at runtime if the actual class of the object cannot be converted legally

Converting and Casting Object References

- Automatic Conversions :

```
class Conversion {
        Fruit f;
        Pineapple p;   // Pineapple extends Fruit

        public void convert () {
            p = new Pineapple ();
            f = p;
        // Compile time error if not cast to Pineapple
            p =(Pineapple) f;
        } }
```

Converting and Casting Object References

- Automatic conversion with interfaces

```
interface Sweet { ... }

class Fruit implements Sweet { ... }
 class MainClass {
Fruit f;
Sweet s;
 public void goodconvert ()
 {
    f = new Fruit();
// legal conversion from class type to  interface type
     s = f;
 }
}
```

Converting and Casting Object References

- Explicit Casts :

```
class AClass {
            void aMethod () { ... }
     }
class BClass extends AClass {
            void bMethod () { ... }
    }
class MainClass {
   public void miscMethod (AClass obj) {
            obj.aMethod ();
            if (obj instanceof BClass)
                          ((BClass)obj).bMethod ();
           }
}
```

Casting with Object references…Examples

$$\begin{tikzcd}
 & X & \\
Y \arrow[ur] & & Z \arrow[ul]
\end{tikzcd}$$

```
X  x = new X();
Y y = new Y();
Z z = new Z();
X xy = new Y();          // compiles ok (up the hierarchy)
X xz = new Z();          // compiles ok (up the hierarchy)
Y yz = new Z();          // incompatible type
Y y1 = new X();          // X is not a Y
Z z1 = new X();          // X is not a Z
X x1 = y;                // compiles ok (y is subclass)
X x2 = z;                // compiles ok (z is subclass)
Y y1 = (Y) x;            // compiles ok but produces runtime error
Z z1 = (Z) x;            // compiles ok but produces runtime error
Y y2 = (Y) x1;           // compiles and runs ok (x1 is type Y)
Z z2 = (Z) x2;           // compiles and runs ok (x2 is type Z)
 Y y3 = (Y) z;           // inconvertible types (casts sideways)
Z z3 = (Z) y;            // inconvertible types (casts sideways)
Object o = z;
```

Object: The Cosmic Superclass

- Every class in Java extends Object
- You can use a variable of type Object to refer to objects of any type:

```
Object obj = new  Employee(“Harry Hacker", 35000);

Employee e = (Employee)obj;
```

Persistent

The toString() Method

- Override toString() to read something meaningful.

O/p :

```
HardToRead@a47e0
```

```
class HardToRead {

        public static void main (String [] args)
        {
                HardToRead h = new HardToRead();
                System.out.println(h);
        }
}
```

The toString() Method

```
class BobTest {
      public static void main (String[] args)
      {
               Bob f = new Bob("GoBobGo");
               System.out.println(f);
       }}
class Bob {

   String nickName;
   Bob(String nickName) {
       this.nickName = nickName;
   }
   public String toString() {
     return ("I am a Bob, but you can call me
"+nickName);

   }}
```

O/p - I am a Bob, but you can call me GoBobGo

Enums

- Enum is a data type that contains fixed set of constants, that specifies the possible values to be used.
- The constants declared in enum are by default static & final.
- E.g. constants like days of week – Sunday, Monday,…..so on

  payment mode – cash, debit card, credit card

Syntax of enum

- The keyword used for declaration is enum.

```
enum Level {
        HIGH,
        MEDIUM,
        LOW
 }
```

Enums

- Enums can be declared inside a class or outside class.
- The example below declares an enum within a class.

```
public class EnumTest{

   public enum Seasons
   {
         SUMMER, WINTER, AUTUMN, FALL
   }
  public static void main(String args[]) {
        //statements to be executed
  }
  }
```

Enums

- The example below shows the enum declared outside class.

```
public enum Seasons
{
  SUMMER, WINTER, AUTUMN, FALL
}

public class EnumTest  {
          public static void main(String args[])
          {
                      Seasons seasons; //usage of enum
                      //Statements to be executed
          }
           }
```

Points to remember about Enums

- Enums:
  - implicitly extend java.lang.Enum class.
  - can not extend any class as implicitly extends java.lang.Enum class.
  - can improve type check
  - can be used in if statements & switch statements.
  - can have fields, methods and constructors.
  - may implement one or more interfaces.
  - can not be instantiated, as constructor is private by default.

How to use Enums?

Here enum is declared outside the class

default & public are valid access modifiers.

```
public enum Seasons{
        SUMMER, WINTER, AUTUMN, FALL
}
public class EnumTest{
        public static void main(String args[]){
        Seasons season; //variable of type Seasons
        Season season = Seasons.WINTER;
        //assigning value to enum variable
        }
}
```

How to use enums?

Here enum is declared inside class

private, default, protected & public are valid access modifiers. Can also be declared as static.

```
public class EnumTest{
        public enum Seasons{
                SUMMER, WINTER, AUTUMN, FALL
        }
        public static void main(String args[]){
                Seasons season; //variable of type
Seasons
                Season season = Seasons.WINTER;
                //assigning value to enum variable
        }
}
```

Enums in if statements

```
public class EnumIfStatementDemo {
        public enum Level{
                HIGH, MEDIUM, LOW;
        }
        public static void main(String[] args) {

        Level waterLevel = Level.MEDIUM;
        if(waterLevel == Level.LOW)
        System.out.println("Water level is low");
        else if(waterLevel == Level.MEDIUM)
        System.out.println("Water level is medium");
        else
        System.out.println("Water level is high");

        }
}
```

Output:
Water level is medium

Enums in switch statement

Output:
Water level is high

```
public class EnumSwitchDemo {
        enum Level{
                HIGH, MEDIUM, LOW;
        }
public static void main(String[] args) {
        Level waterLevel = Level.HIGH;
        switch (waterLevel) {
        case LOW:
        System.out.println("Water level is low");
        break;
        case MEDIUM:
        System.out.println("Water level is medium");
        break;
        case HIGH:
        System.out.println("Water level is high");
        break;
      }  }  }
```

Enum iteration using values() method

- values() method can be used to iterate over the enum & fetch the available values.
- This method returns array of type enum that holds all the values.
- Level levels[] = Level.values();

Persistent

Enum iteration using values() method

```
public class EnumIterationDemo {
        enum Level{
                HIGH, MEDIUM, LOW;
        }

        public static void main(String[] args) {
                Level levels[] = Level.values();

                for (Level level : levels) {
                System.out.println(level);
                }
        }
}
```

```
Output:
        HIGH
        MEDIUM
        LOW
```

Enums with fields & constructor

- The fields can be declared in enum.
- The values for these fields will be passed through constructor while declaring the constants.

```
public enum Level {
                    HIGH(3), MEDIUM(2), LOW(1) ;

                   private int levelCode;

                   Level(int levelCode) {
                             this.levelCode = levelCode;
                    }
          }
```

Enums can have methods

- Methods can be declared in the enums.

```
public enum Level {

        HIGH  (3),  //calls constructor with value 3
        MEDIUM(2),  //calls constructor with value 2
         LOW   (1)   //calls constructor with value 1
        private final int levelCode;

        Level(int levelCode) {
                this.levelCode = levelCode;
        }
        public int getLevelCode() {
                return this.levelCode;
         }
         }
```

Summary :

With this we have come to an end of our session, where we discussed about ….

- Object Oriented Programing concepts in detail
- Inheritance, polymorphism, static keyword
- Reference data member casting
- Enums

At the end of this session, we see that you are now able to answer following questions:

- What are object oriented programming concepts?
- How to implement inheritance, polymorphism?
- What is static keyword and how to use it?
- How to cast reference data members?
- What are enums and how to use enums?

Appendix

References

Thank you

Reference Material : Websites & Blogs

- https://docs.oracle.com/javase/tutorial/java/concepts/
- http://www.javatpoint.com/java-oops-concepts
- http://www.tutorialspoint.com/java/java_overview.htm
- https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html
- http://www.javatpoint.com/enum-in-java

Reference Material : Books

- ***Head First Java***
  - *By: Kathy Sierra, Bert Bates*
  - *Publisher: O'Reilly Media, Inc.*

- ***Java Complete Reference***
  - *By Herbert Schildt*

Persistent

Thank you!

Persistent Interactive | Persistent University